7 UNIX I/O System, I/O Redirection and Piping

Known from Chapter 6, in UNIX, it is through a special file to access one of hardware devices, including character devices (such as the keyboard and printer) and block devices (such as the hard disk). Each hardware device is corresponding to at least one special file. To access a device, use the command or system call that accesses its special file. All I/O devices in the UNIX are treated as files and are accessed as such with the almost same read and write system calls that are used to access all ordinary files (Isaak et al 1998; Jespersen 1995; Sarwar et al 2006). The difference is that device parameters must be set by using a special system call.

Because of the mechanisms that UNIX uses to handle I/O devices, usually, standard files, I/O redirections, and pipes and filters are more relative to the file system in UNIX. In this book, these topics are combined together with I/O devices just to show readers some inherent relationship between I/O devices and the file system.

As we have discussed some concepts about standard files in previous chapters, it is easy for readers to get into I/O system from the standard input and output devices. Thus, we will begin from standard input and output, and standard files in this chapter. Then we will introduce I/O redirections, and pipes, and filters. Finally, I/O system implements in UNIX will be discussed.

7.1 Standard Input and Output, Standard Files

Some UNIX programs or commands read input, some write output, and some do both. Sometimes, the programs or commands read input from a terminal keyboard or write output to a terminal screen; sometimes, they read from a file or write to a file. When a user types in a command on the keyboard, if making some mistakes, such as a command needs a directory argument but the user typing in a file argument, what will happen? Usually, the error is prompted on the terminal screen. However if the command is running in the background and the user does not want the error prompt popping out when

Y. Liu et al., *UNIX Operating System*

doing other jobs, how can the user do to meet the need?

As known in UNIX, devices are treated as special files and those terminal keyboard and screen are I/O devices. As a big interface between users and terminal devices, UNIX file system must be passed through for users to interact with the computer. Hence, what kind of files is associated with the terminal keyboard and screen? In UNIX, there are three standard files where a command reads its input and sends its output and error messages, called standard input, standard output, and standard error, respectively (Sarwar et al 2006). The input, output, and errors of a command can also be redirected to other files by using the redirection facilities in UNIX. In this chapter, we will discuss how to change the input or output of a command between the terminal and a file, and how to use redirection and pipes to combine some commands together to do more complex jobs.

7.1.1 Standard Input and Output

In Section 6.5.6, standard input and standard output have been defined. In this chapter, we can learn them from the I/O system aspect rather than from the file system. Standard input and standard output are I/O devices. So they have their own special files. By default, the terminal keyboard is the standard input, and the terminal screen is the standard output. As running a command is the typical mission for the UNIX shell, the terms of input and output are defined according to a running command. That is, the standard input is where a command receives its input data from while the standard output is the place where a command sends its output data to. It is also helpful for readers to understand the redirection and pipe concepts in the rest of this chapter.

7.1.2 Standard Input, Output and Error Files

When UNIX commands are executed, the kernel opens three standard files: stdin is the standard input file, stdout is the standard output file, and stderr is the standard error file. By default, each of the commands takes its input from the standard input and sends the results to the standard output. These standard files are attached to the terminal where the user types in commands. When a command is running on the UNIX shell, by default, the command input comes from the terminal keyboard and its output and error messages go to the terminal screen (or a terminal window). The command reads input from stdin file and sends its output and error messages to stdout and stderr files, respectively. The illustrative relationships between standard I/O and standard files when running a command are shown in Figure 7.1.

Mentioned in Section 6.5, every open file in UNIX file system has a file descriptor. And standard files have their file descriptors, too. They are 0 for stdin, 1 for stdout, and 2 for stderr. Figure 7.2 shows the logical relationship of the file descriptors, standard files and standard I/O.

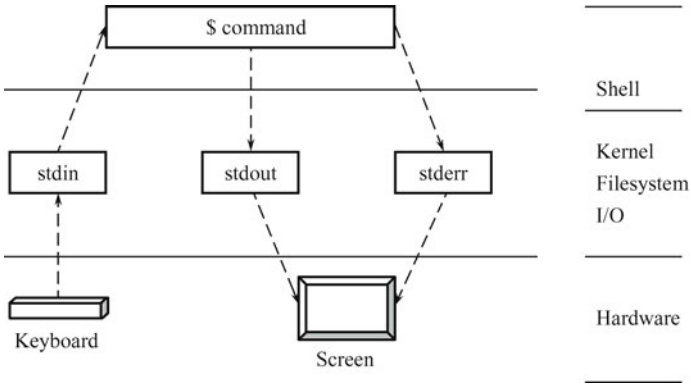

Fig. 7.1 Illustrative relationships between standard files and standard I/O in command default execution.

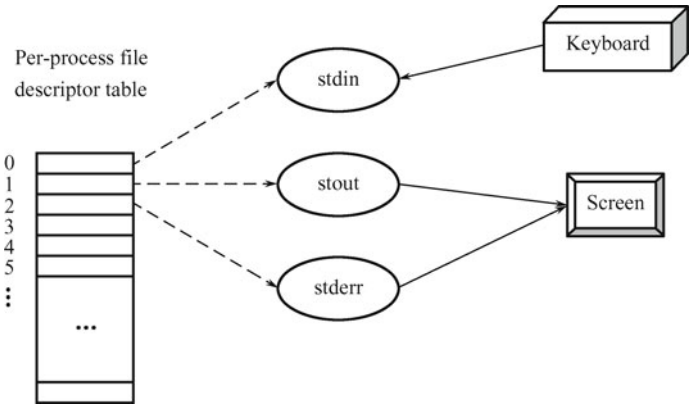

Fig. 7.2 The logical relationship of the file descriptors, standard files and standard I/O.

7.2 Input Redirection

In UNIX, input redirection means to let a command input direct to a file rather than to standard input – a terminal keyboard. So once done, the redirected command receives the input from the specified file not from the terminal keyboard. There are two different ways to do input redirection. One

is to use the less-than symbol (<); the other is to use the < combined with the file descriptor of the standard input file, stdin. They will be discussed, respectively. The effect of the input redirection is illustrated in Figure 7.3.

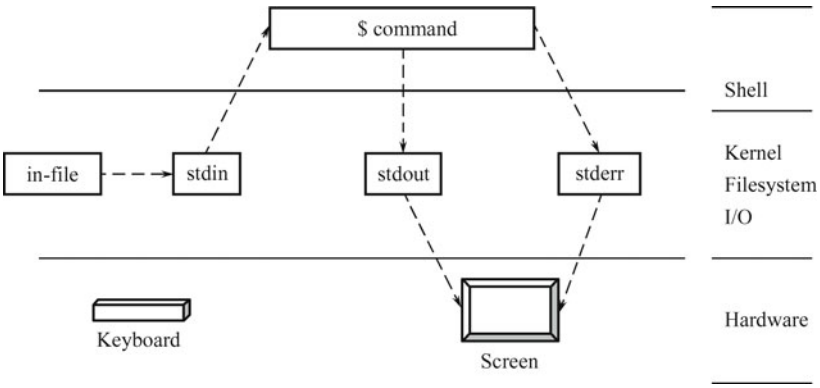

Fig. 7.3 Illustrative relationships between standard files and standard I/O when an input-redirected command executing.

7.2.1 Input Redirection with < Operator

The syntax and function of the input redirection operator < are as follows.

```
$ command < in-file
```

Function: to make the command input come from the in-file rather than the terminal keyboard.

Note: The command, here, should need the input from the keyboard by default. That is, if a command does not need an input naturally, the input redirection operator in the command makes no sense.

For example,

```
$ grep "Hill" < freshman
Miller Hill     1322223434      millerhill8806@sina.com
Gao Hill        1361010101      gaoyuling@hebust.edu.cn
$
```

As known in Chapter 6, when using grep command without any argument, grep takes input from the keyboard. This command is different from the command grep Hill freshman in Section 6.3.6. This command is without argument and takes the file freshman as input, but the command grep Hill freshman has an argument freshman. Without other redirections, the output of this grep command is sent to the standard output – the screen.

Another example is as follows.

```
$ cat < file6
```

```
...
$
```

Known from the previous example, this cat command takes its input from the file file6 and displays the content of file6 on the screen. Even though the result is the same as the command cat file6, two commands are different.

7.2.2 Input Redirection with File Descriptor

Described above, the file descriptor of the standard input file is 0. This integer number can be used to redirect a command input. Most of UNIX shells allow to use file descriptors to open files and associate file descriptors, but the C shell does not use file descriptors with redirection operators. Thus, standard input can be redirected by using the 0< operator in most shells but not in C shell.

The syntax and function of the 0< operator are as follows.

```
$ command 0< in-file
```

Function: to make the command input come from the in-file rather than the terminal keyboard.

Note: The command, here, should also need the input from the keyboard by default.

Two examples in the previous section can be written down as the following and their effects are the same as the previous two.

```
$ grep "Hill" 0< freshman
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
$ cat 0< file6
...
$
```

As these two commands have the same functions as the ones in the previous section, the previous are simpler, but these two are clearer.

7.3 Output Redirection

As input redirection, output redirection means to let a command output direct to a file rather than to standard output — a terminal screen. So once done, the redirected command sends the output to the specified file not to the terminal screen. There are also two different ways to do output redirection. One is to use the greater-than symbol (>); the other is to use the > combined with the file descriptor of the standard output file, stdout. They will be discussed, respectively. The effect of the output redirection is illustrated in Figure 7.4.

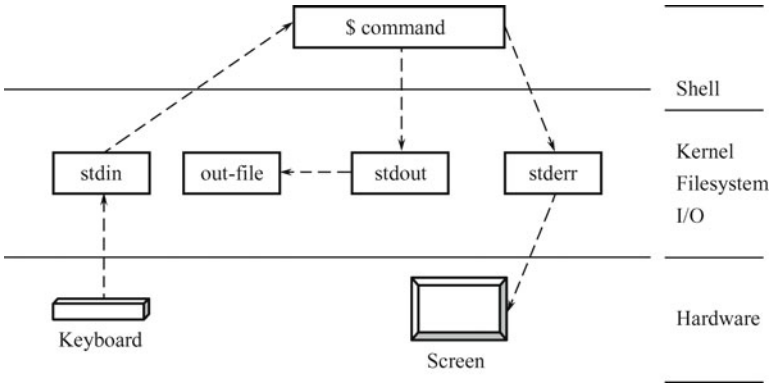

Fig. 7.4 Illustrative relationships between standard files and standard I/O when an output-redirected command executing.

7.3.1 Output Redirection with > Operator

The syntax and function of the output redirection operator > are as follows.

```
$ command > out-file
```

Function: to make the command output go to the out-file rather than the terminal screen.

For example,

```
$ grep Hill freshman > gfreshman
$ cat < gfreshman
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
$
```

With the output redirection, the output of this grep command is sent to the gfreshman file. So the result cannot be seen on the screen. But it can be displayed on the screen by using the cat command as shown.

Another interesting example is as follows.

```
$ cat > file7
... typing the content of file7
CTRL-D
$
```

Remember that the cat command sends its output to standard output – the display screen by default. The standard input of this command is still the keyboard. Therefore, when this command is executed, it creates a file called file7 and waits for the user to type in on the keyboard. When finishing the text of file7, press Ctrl-D in the first column of a new line to quit the cat command. If the file7 exists, by default it will be overwritten. Now, another

way to create a new file has been learned.

7.3.2 Creating a File with Output Redirection

Many methods to create a file have been learned in the previous chapters, such as using a text editor, using the copy command, etc. In this section, a new method to create a small text file with output redirection will be introduced in detail.

Assuming that a file for a work diary, named td-workdiary is needed to create, the steps are given as follows.

```
$ cat > td-workdiary
Read computer books 30 pages in the morning
Do the science research in laboratory in the afternoon
Write work notes in the evening
CTRL-D
$
```

The first part is to create the td-workdiary file by using the cat command without an argument and with output redirection to the td-workdiary file. The cat command without an argument takes the input from the keyboard. Type in some lines of text and terminate the command by pressing CTRL-D on a new line. Then continue the following steps.

```
$ date > td-time
$
```

The second part is to make the td-time file store the time by using the date command with output redirection to the td-time file.

```
$ cat td-time td-workdiary > myworkdiary
$ cat myworkdiary
Tues Aug 19 08:04:56 GMT 2008
Read computer books 30 pages in the morning
Do the science research in laboratory in the afternoon
Write work notes in the evening.
$
```

The third part is first to create the myworkdiary file by using the cat command with two arguments (td-time and td-workdiary) and with output redirection to the myworkdiary file. Then display the content of the myworkdiary file by using the cat command. Now, the work diary, named myworkdiary, is created.

7.3.3 Output Redirection with File Descriptor

Described above, the file descriptor of the standard output file is 1. This integer number can be used to redirect a command output. By using file

descriptor, standard output can be redirected by using the 1> operator in most shells.

The syntax and function of the 1> operator are as follows.

```
$ command 1> out-file
```

Function: to make the command output go to the out-file rather than the terminal screen.

Two examples in the previous section can be rewritten down as the following and their effects are the same as the previous two.

```
$ grep Hill freshman 1> gfreshman
$ cat 0< gfreshman
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
$
$ cat 1> file7
... typing the content of file7
CTRL-D
$
```

As input redirection, the previous two commands are simpler, but these two are more obvious.

7.4 Appending Output Redirection

By default, output and error redirections overwrite contents of the specified files. If a user does not expect to overwrite the previous contents of the destination files, UNIX provides a solution to deal with this problem — by using the appending redirection operator >>.

7.4.1 Appending Output Redirection with >> Operator

The syntax and function of the >> operator are as follows.

```
$ command >> out-file
```

Function: to make the command output go to the out-file rather than the terminal screen and append the new content to the end of the out-file if it is existent.

For example,

```
$ grep Hill freshman >> gfreshman
$ cat 0< gfreshman
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
$
```

The result of the grep command is interesting. In the gfreshman file, there are two same lines for "Miller Hill" and "Gao Hill". That result is assuming that this grep command is done after the previous grep command in the last section. This grep command does not overwrite the previous content of the gfreshman file but appends the new contents at the end of the file.

7.4.2 Appending Output Redirection with the File Descriptor

The syntax and function of the 1>> operator are as follows.

```
$ command 1>> out-file
```

Function: to make the command output go to the out-file rather than the terminal screen and append the new content to the end of the out-file if it is existent.

Note: As the > operator, the default effect of the >> operator is the output redirection. So the result of a command with the 1>> operator is equivalent to the one with the >> operator. But file descriptor 2 can be used to append errors to a file, which will be discussed later in this chapter.

The example in the previous section can be rewritten down as follows and their effects are the same.

```
$ grep Hill freshman 1>> gfreshman
$ cat 0< gfreshman
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

7.5 Standard Error Redirection

As output redirection, standard error redirection means to let the error caused by the command execution direct to a file rather than to standard output – a terminal screen. Once done, the redirected command sends the execution error to the specified file not to the terminal screen. As the standard error file and standard output file are both associated with the terminal screen by default, the standard error is redirected only by using the > operator combined with the file descriptor 2 of the standard error file, stderr. The effect of the standard error redirection is illustrated in Figure 7.5.

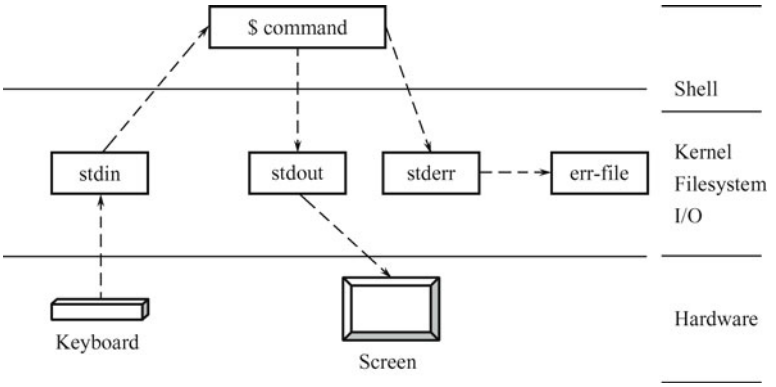

Fig. 7.5 Illustrative relationships between standard files and standard I/O when an error-redirected command executing.

7.5.1 Error Redirection by Using File Descriptor

As mentioned above, most of UNIX shells allow to use file descriptors to open files and associate file descriptors, but the C shell does not use file descriptors with redirection operators. The standard error redirection in the C shell will be discussed in the next section. In most UNIX shells, standard error output can be redirected by using the 2> operator.

The syntax and function of the 2> operator are as follows.

```
$ command 2> err-file
```

Function: to make the command execution error go to the err-file rather than the terminal screen.

For example,

```
$ grep Hill freshman 2> efreshman
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

Without the output redirection, the output of this grep command is sent to the terminal screen. If efreshman does not exist, it is created; otherwise, it is overwritten.

Another example,

```
$ sort encyclopedia > encyclopedia.st 2> /dev/null &
[3] 18801
$
```

Remember this is the example in Section 4.5.1. This command sorted a huge file named encyclopedia, sent the sorting result into the file encyclopedia.st, and would send the error messages into the file /dev/null if there

were errors while the command running. And 2> /dev/null means make the standard error redirect to the file /dev/null. The /dev/null is an interesting file, and anything goes into it will disappear. This command is running at the background.

However, sometimes, it is useful to keep the standard error attached to the display screen because it can prompt the user instantly if the command is not typed in a correct way, such as typing a nonexistent directory as a command argument, or the user without the read permission for a directory.

7.5.2 Appending Error Redirection by Using File Descriptor

As the 1>> operator, the 2>> operator can avoid to overwrite an existent file.

The syntax and function of the 2>> operator are as follows.

```
$ command 2>> err-file
```

Function: to make the command execution error go to the err-file rather than the terminal screen and append the new content to the end of the err-file if it is existent.

For example,

```
$ grep Hill freshman 2>> efreshman
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

The output of this grep command is sent to the terminal screen. If efreshman does not exist, it is created; otherwise, it is appended to its end with the new error messages of this command execution.

7.5.3 Error Redirection in C Shell

In the C shell, the input, output, and appending redirection operators (<, >, >>) do work as well as they do in other shells. But these operators combined with file descriptors do not function well in the C shell. In the C shell, there is not an operator to redirect the stderr alone. But the >& operator can be used for both the output and error redirections at the same time.

The syntax and function of the >& operator is as follows.

```
% command >& outerr-file
```

Function: to make the command output and error messages go to the outerr-file rather than the terminal screen.

```
For example,
```

```
% ls − l >& detailandeofd
%
```

This ls command redirects its output and error messages to the detailandeofd file. Note: % sign is the C shell prompt, just like $ is the Bourn shell prompt.

Even though the C shell does not have an operator just for error redirection, the C shell solution is to use the redirection operators in parentheses so that the command can be executed in different subshells (see Section 4.5.2). For example,

```
% (sort encyclopedia > encyclopedia.st ) >& err-sort &
%
```

As mentioned in Chapter 5, when an external command is executed on the shell, the kernel forks a subshell that inherits the standard files of the parent shell and executes the command. The stdout and stderr of the parent shell are redirected to the err-sort first, and then the subshell that is created for a sort in parenteses is redirected the stdout to the encyclopedia.st. Therefore, the command in the parentheses sends its output to the encyclopedia.st file and remains the error messages go into the err-sort file.

In the C shell, the >>& operator can be also used. As the >> operator, the >>& operator can avoid to overwrite an existent file.

The syntax and function of the >>& operator is as follows.

```
% command >>& outerr-file
```

Function: to make the command output and execution error go to the out-file rather than the terminal screen and append the new contents to the end of the out-file if it is existent.

For example,

```
% ls −l >>& detailandeofd
%
```

This ls command appends its output and error messages to the detailandeofd file.

7.6 Combining Several Redirection Operators in One Command Line

In the most UNIX shells, some redirection operators can also be used in one command line. Some combinations will be discussed.

7.6.1 Combining Input and Output Redirections in One Command Line

Input and output redirection operators can be used together by combining in one command. When both input and output operators are used in one command, the effect of the redirected command is illustrated in Figure 7.6.

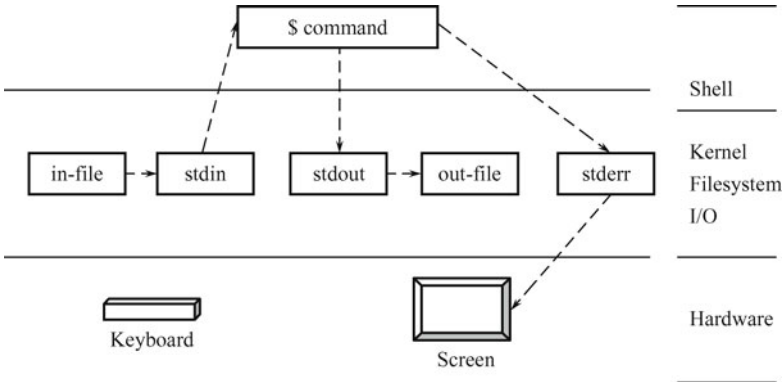

Fig. 7.6 Illustrative relationships between standard files and standard I/O when an input-and-output-redirected command executing.

The syntax and function of combining the input and output redirection operators are as follows.

```
$ command < in-file > out-file
$ command > out-file < in-file
$ command 0< in-file 1> out-file
$ command 1> out-file 0< in-file
```

Function: to make the command input come from the in-file rather than the terminal keyboard and the command output go to the out-file than the terminal screen.

For example,

```
$ grep "Hill" < freshman > gfreshman
$
```

This command is without argument, takes the file freshman as input, and sends its output to the gfreshman file. If the gfreshman file exists, overwrites it; if not, creates it.

Another example is as follows:

```
$ cat < file6 > file7
$
```

This cat command takes its input from the file file6 and sends the content of file6 to the file7 file. Its function seems like the command cp file6 file7, but two commands are slightly different. If the file7 exists, the cat < file6 >file7 command first empties the file7 file and copies the contents of file6 into it.

In this way, the file7 file still has the original file attributes, such as the user permissions for the file. But the cp file6 file7 command makes the file7 file from the inode copy. So the file7 and file6 files are totally the same, not only in their contents but in their attributes. If the file7 does not exist, the two commands will have the same file7.

7.6.2 Combining Output and Error Redirections in One Command Line

The output and error messages of a command can be redirected in one command line.

The syntax and function of combining the output and error redirection operators are as follows.

```
$ command 1> out-file 2> err-file
$ command 2> err-file 1> out-file
```

Function: to make the command output and error messages go to the out-file and err-file, respectively, rather than the terminal screen.

Note: Both of two given commands function the same, and the effect of them is shown in Figure 7.7.

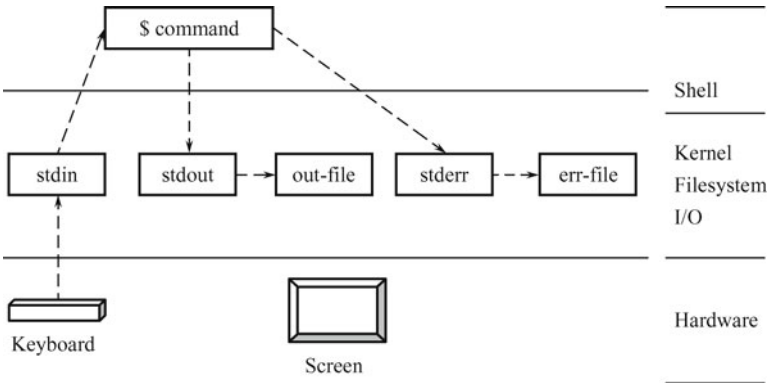

Fig. 7.7 Illustrative relationships between standard files and standard I/O when an output-and-error-redirected command executing.

For example,

```
$ grep "Hill" freshman 1> gfreshman 2> efreshman
$
```

The output of this grep command is sent to the gfreshman file and its error messages are sent to the efreshman file. If gfreshman and efreshman don't exist, they are created; otherwise, overwritten.

To put the output and error messages of a command into one file, use the

following command syntax.

```
$ command 1> outerr-file 2>&1
$ command 2> outerr-file 1>&2
```

Function: to make the command output and error messages go to the same file of outerr-file rather than the terminal screen.

Note: Both of two given commands have the same function – to put the command output and error messages into one file. But the 2>&1 operator in the first command means to copy descriptor 1 to descriptor 2, and the 1>&2 operator in the second command means to copy descriptor 2 to descriptor 1. Usually a UNIX shell interprets and executes a command from left to right. Therefore, if one term is dependent on another in a command, be careful with this rule. In this case, as one descriptor is a copy of another, redirections must be specified in left-to-right order. That is, it is necessary to let the copy source goes before the copy destination. In other words, to use 2>&1, it is necessary to let the 1> operator goes first. The effect of them is shown in Figure 7.8.

For example:

```
$ grep "Hill" freshman 1> gefreshman 2>&1
$
```

or

```
$ grep "Hill" freshman 2> gefreshman 1>&2
$
```

These two commands send the output and error messages into the same gefreshman file, if some error produced.

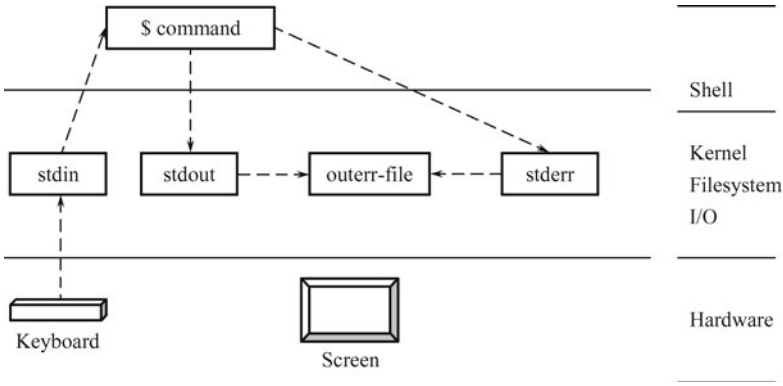

Fig. 7.8 Illustrative relationships between standard files and standard I/O when another output-and-error-redirected command executing.

7.6.3 Combining Input, Output and Error Redirections in One Command Line

The input, output and error messages of a command can also be redirected in one command line.

The syntax and function of combining the input, output and error redirection operators are as follows.

```
$ command 0< in-file 1> out-file 2> err-file
```

Function: to make the command input from the in-file file rather than from the terminal keyboard, and make the command output and error messages go to the out-file and err-file, respectively, rather than the terminal screen.

Note: In this syntax, the order of the redirection operators is also important. If the in-file does not exist, the error message will be sent to the terminal screen because the error redirection has not been done. The effect of this syntax is shown in Figure 7.9.

For example:

```
$ grep "Hill" 0> freshman 1> gfreshman 2> efreshman
$
```

This command gets its input from the freshman file, sends the output and error messages into the gefreshman file and efreshman file, respectively, if some error produced.

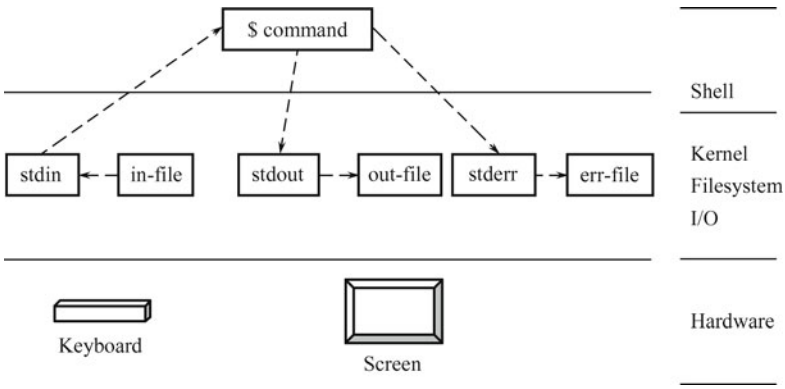

Fig. 7.9 Illustrative relationships between standard files and standard I/O when an input-output-and-error-redirected command executing.

7.6.4 Combining Appending Redirection with Other Redirections in One Command Line

The appending redirection >> operators can also be combined together with themselves or other redirection operators in one command line. In this way, it can avoid overwriting some files by accident. The following are given some of examples.

```
$ grep Hill freshman 1>> gfreshman 2>> efreshman
$
```

This grep command appends its output to the gfreshman file and its error messages to the efreshman file.

```
$ ls -la 1>> lsoutput 2> lserror
$
```

This ls command appends its output to the lsoutput file and sends its error messages to the lserror file. If the lserror file does not exist, create it; if it exists, overwrite it.

```
$ cat file1 file2 file3 >> file4 2> cerror
$
```

This cat command adds file1, file2, and file3 to the end of the file4 file. The file1, file2, and file3 files are the input of the command. The file4 file is the output destination of the command. And the error messages are redirected to the cerror file. If the cerror file does not exist, create it; if it exists, overwrite it.

In the C shell and Korn shell, it can also avoid overwriting an existent file to set the noclobber option. If putting this set command in the. profile or. cshrc, the noclobber option can be set permanently. The. profile file has been discussed in Chapter 2.

If the noclobber variable is set, the command cat filel file2 file3 > file4 generates an error message if file4 exists. If file4 does not exist, it is created and the files of filel, file2, and file3 are copied into it. When noelobber is set, the command cat filel file2 file3 >> file4 functions well if file4 exists while an error message is generated if file4 does not exist.

However, the >!, >>!, and >>&! (each operator with an exclamation mark) operators can override the effect of the set noelobber variable. Therefore, even if the noclobber variable is set and file4 exists, the command cat filel file2 file3 >! file4 copies the files of filel, file2, and file3 into it.

7.7 UNIX Pipes and Filters

As mentioned above, the input or output of a command can be redirected to a file by using the input or output redirection operators. It is also possible in UNIX to connect the output from one command with the input to another

command. Hence, the output of the first command becomes the input of the second command. In other words, the UNIX allows the stdout of one command to be connected to the stdin of another command.

7.7.1 Concepts of Pipe and Filter

To make a connection, the pipe operator (a vertical bar |) can be used. The syntax of the pipe operator is as follows.

```
$ command1 | command2 | command3
```

Function: to make the command1 output connect to the command2 input, and the command 2 output connect to the command3 input.

Note: In fact, the command list can go on and on for several commands in one command line if the connections between each pair of all the commands are meaningful. The effect of this syntax is shown in Figure 7.10.

When a pipe is set up between two commands, the standard output of the command to the left of the pipe operator becomes the standard input of the command to the right of the pipe operator. Any two commands can form a pipe as long as the first command writes to standard output, and the second command reads from standard input.

If a command takes its input from another command, does some operation on that input, and writes the result to the standard output, which may be piped to yet another command, the command is called a filter.

Most UNIX commands can be used to form pipes. In the following sections, some commands that can be used as filters will be discussed.

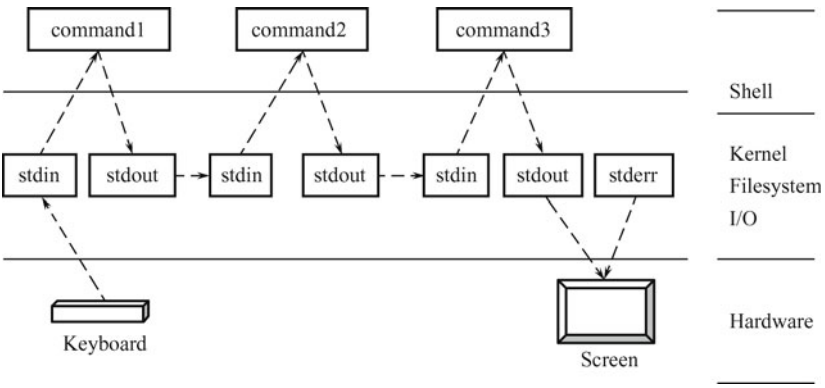

Fig. 7.10 Illustrative relationships between standard files and standard I/O when piped commands executing.

7.7.2 Examples of Pipes and Filters

In UNIX, pipes and filters are usually used to perform some complicated tasks that cannot be done with a single command. Commonly, some commands, such as cat, grep, lp, pr, sort, tr, and wc, can be used as filters.

For example,

```
$ ls -la
Total 32
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   .
drwxr-xr-x    2   admi   admi      512    Apr   12   15:01   ..
-rw-r--r--    2   wang   project   136    Jan   16   11:48   .exrc
-rw-r--r--    2   wang   project   833    Jan   16   14:51   .profile
-rw-r--r--    1   wang   project   797    Jan   16   15:02   file1
-rw-r--r--    1   wang   project   251    Jan   16   15:03   file2
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   text
...
$ ls -la | grep "wang"
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   .
-rw-r--r--    2   wang   project   136    Jan   16   11:48   .exrc
-rw-r--r--    2   wang   project   833    Jan   16   14:51   .profile
-rw-r--r--    1   wang   project   797    Jan   16   15:02   file1
-rw-r--r--    1   wang   project   251    Jan   16   15:03   file2
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   text
...
$
```

The above command line executes ls -la command to list the working directory. The output is piped to the grep "wang" command that only displays on the terminal screen the lines that contain the string wang, that is, the files whose owner is wang.

Second example:

```
$ ls -la | more
......
$
```

The difference between this command line and the ls -la command is that this command line can display the information of the working directory one screen at a time because the more command is a pager. If the list of the directory information is so long that it covers several screens, this command line will be helpful to see the detailed information one screen by one screen.

7.7.3 Combining Pipes and I/O Redirections in One Command Line

Two or more commands can be connected with pipes alone. And also, pipes and I/O redirections can be used in one command line. The related examples will be given in the next section. But it is impossible to realize to redirect the output of a command to a file and connect it to the input of another

command at the same time by just using pipes and I/O redirections in one command line. But it can be done with the help of the tee operator. Its syntax is as follows:

```
$ command1 | tee file1 file2 file3 |command2
```

Function: to make the command1 output connect to the tee input, and the tee command send its input to the files file1 through file3 and to the command3 input.

Note: The tee operator tells the shell to send the output of a command to one or more files, as well as to another command at the same time. And the file list after the tee operator can go on and on for several files in one command line. The effect of this syntax is shown in Figure 7.11.

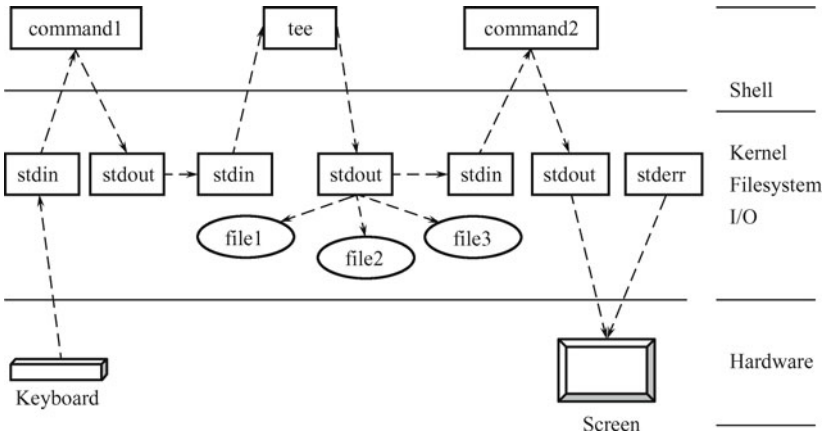

Fig. 7.11 Illustrative relationships between standard files and standard I/O when executing commands with tee.

The command is executed and its output is stored in the files of file1, file2, and file3, and sent to command2 as its input. The following is an example.

```
$ grep -v "David" freshman | tee fresh1 fresh2 | sort +1 -2 -b > fresh3
$
```

In this command line, the output of the grep -v “David” freshman command is piped to the input of tee, which puts copies of these lines in the files of fresh1 and fresh2, and sends them to the input of the sort +1 -2 -b > fresh3 command. Finally, the result of the sort command is sent to the fresh3 file. Thus, those lines in the freshman file that do not contain David are saved in the files of fresh1 and fresh2. If later on there are two different contents for the two files, respectively, it is useful to have these two copies.

7.7.4 Practical Examples of Pipes

In the examples of the previous section, pipes are implemented in the main memory but not on disk, which is an I/O device. To see the difference clearly, here give another example that is equivalent to the second example of Section 7.7.2.

```
$ ls −la > rsofls
$ more rsofls
......
$
```

Known before, the I/O access can slow down significantly the execution of the commands by using the read and write system calls for the rsofls file on disk. But the real situation is not that bad for the I/O access because of the cache buffer for the disk access of the UNIX kernel.

Second example: Here, by using the example of Section 6.3.5 in Chapter 6 as another example, do the following process.

```
$ cat freshman
Jones David          1358999998      jonesdavid@hebust.edu.cn
Huo Song             1341212121      huosong88@126.com
Dang Ailin           1500011223      dangailin@163.com
Jones David          1357777888      jdavid87@sina.com
Liang Fanghua        1310000555      liangfh88@163.com
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Nan Xee              1393219123      nanxee1987@aaa.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
$ cat freshman | grep -v "David" | sort +1 -3 -b
Dang Ailin           1500011223      dangailin@163.com
Liang Fanghua        1310000555      liangfh88@163.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Huo Song             1341212121      huosong88@126.com
Nan Xee              1393219123      nanxee1987@aaa.com
$
```

The first cat freshman command displays the lines in the freshman file. In the second command line, the output of the cat freshman command is sent to the input of the grep -v “David”, which should display those lines that do not contain David in the freshman file; then the output of the grep -v “David” is piped to the input of the sort +1 -3 -b command, which does sorting according to the first sort key (+1 -3), given name (Field 1) and cell-phone number, and displays the result on the screen.

Now, the power of the pipe can be seen, which is to perform the functions of three commands in one command line by forming a pipeline of commands and is equivalent to the following command sequence.

```
$ cat freshman > tempfresh1
$ grep -v "David" tempfresh1 > tempfresh2
$ sort +1 -3 −b tempfresh2
$ rm tempfresh1 tempfresh2
```

```
$
```

The command line with pipes does not need any disk file as a temporary storage place for processing the data, but the command sequence uses two temporary disk files and four or more disk I/O (read and write) operations depending on the file system.

Third example: I/O redirection and pipes can be used in one command line. Here is the previous example with different processes.

```
$ cat freshman
Jones David          1358999998      jonesdavid@hebust.edu.cn
Huo Song             1341212121      huosong88@126.com
Dang Ailin           1500011223      dangailin@163.com
Jones David          1357777888      jdavid87@sina.com
Liang Fanghua        1310000555      liangfh88@163.com
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Nan Xee              1393219123      nanxee1987@aaa.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
$ cat < freshman | grep -v "David"
Huo Song             1341212121      huosong88@126.com
Dang Ailin           1500011223      dangailin@163.com
Liang Fanghua        1310000555      liangfh88@163.com
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Nan Xee              1393219123      nanxee1987@aaa.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
$
```

In the second command line, the input of the cat command is from the freshman file and the output of the cat < freshman command is piped to the input of the grep -v "David", which displays on the screen those lines that do not contain David in the freshman file.

Forth example: Here is also the previous example with different processes.

```
$ cat < freshman | grep -v "David" | sort +1 -3 -b > stfreshman
$
```

In this command line, the input of the cat command is from the freshman file and the output of the cat < freshman command is piped to the input of the grep -v "David"; then the output of the grep -v "David" is piped to the input of the sort +1 -3 -b command, which redirects its output to the stfreshman file.

7.7.5 Pipes in C Shell

In the C shell, the | operator can connect the output of one command to the input of another one. Otherwise, it also allows the output and error messages of one command to be attached to the input of another command with the |& operator. Its syntax is displayed as follows.

```
% command1 | command2
```

Function: to connect the command1's output to the command2's input.

```
% command1 |& command2
```

Function: to let the command1's output and error messages sent to the command2 as its input.

For example:

```
% cat freshman | sort -1 +2 -b > sfreshman
%
```

The output of the cat command is attached to the input of the sort command. Hence, the output of the cat command is sent to the sort command as its input.

Another example:

```
% cat freshman |& sort -1 +2 -b > sfreshman
%
```

The output and error messages of the cat command are attached to the input of the sort command. Hence, the sort command reads the output of the cat command, or any error produced by its executing, such as, if freshman does not exist.

Third example:

```
% cat freshman | grep -v "David" |& sort +1 -2 -b >stfreshman
%
```

In this command line, the output of the cat command is sent to the input of the grep command. Then, the output and error messages of the grep command are piped to the sort command as its input.

7.7.6 Named Pipes

For the kernel, a pipe is an area in the kernel memory that allows two processes, which are running on the same computer system and related to each other, to communicate with each other. This has been discussed in Section 6.2.5. Thus, a pipe can be used in an inter-process communication.

The pipe differs from a regular file in that the data in a pipe is temporary: once data is read from a pipe, it cannot be read again. Also, the data is read in the order that it was written to the pipe. In other words, the pipe communication is one-way. For example, in the command line of ls -la | grep "wang", the output of ls -la is read by grep "wang" as input. Here, the one-way communication is from ls to grep. For a bidirectional communication between processes, two pipes are needed. This cannot be performed at the shell level, but can be done by using the pipe system call in the C or C++ programming.

A named pipe is a file that allows two processes to communicate with each other if the processes are on the same computer, but do not have to be

related to each other.

The kernel stores data in a named pipe in the same way it stores data in an ordinary file, except that it uses only the direct address, not the indirect address.

Processes that are communicating with pipes usually have the same parent process while processes that are communicating with named pipes can be independently on one system.

The UNIX operating system provides the mkfifo command to create named pipes. Its syntax is as follows.

```
$ mkfifo [option] file[s]
```

Function: to create named pipes; file[s] can be several files with pathnames.

Common options:

-m mode: to set access permissions for named pipes to 'mode'; the 'mode' is set in the same way as with the chmod command (see Section 9.5.1), such as 777 for giving all users the read, write and execute permissions to the created named pipes.

Note: As the mkfifo command can create the named pipes alone, it means the commands that use the named pipes do not necessarily exist and execute in one command line. A named pipe is on disk like a file with a filename. Hence, it can be used like a file, such as accepting the operations of the open, close, read, and write system calls. The execution of the mkfifo command causes the mknod system call that can be used to create a new special file, directory, or named pipe.

For example:

```
$ mkfifo firstfifo
$ mkfifo —m 666 secondfifo
$ ls -l
Total 42
drwxr-xr-x  2  wang   project   512   Apr   15   15:11   .
drwxr-xr-x  2  admi   admi      512   Apr   12   15:01   ..
...
prw-r--r--  1  wang   project   0     Jan   18   15:02   firstfifo
prw-rw-rw-  1  wang   project   0     Jan   18   15:03   secondfifo
...
$
```

The first mkfifo command has created a named pipe, called firstfifo, with default permissions. The second command has created a named pipe, called secondfifo, with read and write permissions for all the users and execute permission for nobody. The ls -l command has displayed the access permissions of the two named pipes.

When two commands use the named pipe called firstfifo to communicate with each other, they can run separately as the following sequence:

```
$ command1 < firstfifo &
$ command2 > firstfifo
```

Note: The first command is put in the background so that the shell

prompt can be appear on the screen again and the second command can be typed in. When the first command is typed in, it is blocked because the fistfifo is empty. When the output of the second command is sent to the firstfifo, the first command starts to read from the firstfifo and process the data in it. The output of the first command displays on the screen. The effect of the named pipe is shown in Figure 7.12.

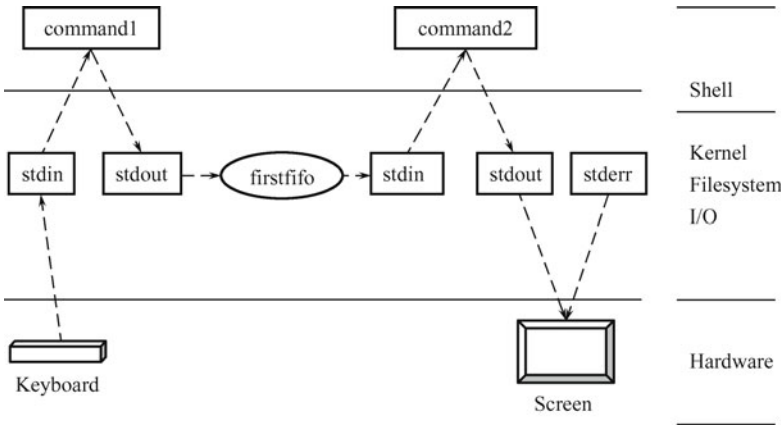

Fig. 7.12 Illustrative relationships between standard files and standard I/O when executing commands with a named pipe.

For example:

```
$ cat firstfifo &
$ more secondfifo &
$ grep -v "David" freshman |tee firstfifo secondfifo | sort +1 -3 -b
......
$
```

The first two commands, cat and more, are blocked until the firstfifo and secondfifo are written. The grep command sends those lines that do not contain David in the freshman file to the tee, which redirects its output to the two named pipes as well as sends it to the sort command. Thus, the sort, cat, and more commands display their outputs on the screen, respectively. As the scheduling of the processes in the system is dependent on the kernel, the outputs of the commands may not appear on the screen in an expected order, but all of their outputs will display on the screen finally.

When a named pipe is no longer needed, it can be removed by the way to remove an ordinary file — using the rm command to remove it from the file system. For example, to remove the firstfifo and secondfifo, use the following command:

```
$ rm firstfifo secondfifo
$
```

If the ls command is used to check, the two named pipes will disappear from the list.

7.8 UNIX Redirection and Pipe Summary

In fact, many commands can be combined together with pipes and redirection operators to perform some complex functions. For readers, it needs some time and practices to know how powerful the pipes and redirection operators are. And also, it is not just for those commands given in this chapter that can combine with pipes and redirection operators. In UNIX, there are some more, especially in the networks.

In this section, a summary of UNIX redirection and pipe operators in the Bourne and C shells is given in Table 7.1. As in the Korn shell, most of the redirection and pipe operators are the same as in the Bourne shell, they are not shown in the table.

Table 7.1 Important redirection and pipe operators in Bourne and C shells

Operator	Use in Bourne Shell	Use in C Shell
< file	Input redirection	Input redirection
0< file	Input redirection	
> file	Output redirection	Output redirection
1> file	Output redirection	
>> file	Append standard output to the file	Append standard output to the file
1>> file	Append standard output to the file	
2> file	Error message redirection	
2>> file	Append standard error to the file	
>& file		Output and error message redirection
>>& file		Append output and error message to the file
m>& n	Copy file descriptor n to file descriptor m	
>! file		Output redirection with ignoring noclobber
>>! file		Append standard output to the file with ignoring noblobber; if the file is inexistent, create it
>>& ! file		Append standard output and error message to the file with ignoring noblobber; if the file is inexistent, create it
comm1 | comm2	Connect the output of the comm1 command to the input of the comm2 command	Connect the output of the comm1 command to the input of the comm2 command

Continued

Operator	Use in Bourne Shell	Use in C Shell
comm1 | & comm2		Connect the output and error message of the comm1 command to the input of the comm2 command

7.9 I/O System Implementation in UNIX

In a computer system, there are peripheral devices attached to it, such as disks, terminals, keyboards, printers, and networks. The UNIX kernel modules that control devices are known as device drivers.

7.9.1 I/O Mechanisms in UNIX

In UNIX, devices are divided into two types, block devices and character devices (Bach 2006; Nelson et al 1996). Block devices, such as disks, are random access storage devices. Character devices include all other devices such as terminals and network media.

A block special file consists of a sequence of numbered blocks. The key property of the block special file is that every block can be accessed with its own address. In other words, a process can open a block special file and read, for example, block 1024 without going through blocks 0 to 1023 first. That is also the meaning of random access storage.

Character special files are usually used for devices with which human beings interact with the computer system directly. Compared to computers, human beings act far slower. Bridging between human beings and computers, these devices input or output a character stream. Keyboards, printer, networks, mice, and most of the other I/O devices belong to these devices.

In UNIX, these I/O devices are integrated into the file system, so the user interface to devices needs to go through the file system to control the I/O devices. Every device has a name that looks like a filename and is accessed like a file. The device special file also has an inode and occupies a node in the directory tree of the file system. The special file is different from regular files by the file type stored in its inode, either "block special" or "character special", corresponding to device it represents.

In the UNIX file system, each I/O device is assigned a pathname, usually in /dev. For example, a disk may be /dev/hd1, a printer may be /dev/lp, a terminal may be /dev/tty2, and the network may be /dev/net.

System calls for regular files, such as open, close, read, and write, have an appropriate meaning for devices. Processes can open, read, and write

special files in the same way as they do regular files. So for users, no special mechanism is needed for doing I/O.

Of course, each device driver does not necessarily support every system call interface. I/O devices have their own system calls that are not applicable to regular files. For example, the system calls of enabling and disabling of character echoing, and conversion between carriage return and line feed are specified for some character special files.

In UNIX, I/O is operated by a set of device drivers. The function of the drivers is to isolate the rest of the system from the specific controls on the hardware. By providing standard interfaces between the drivers and the rest of the operating system, most of the I/O system can be put into the machine-independent part of the kernel, which is important for operating system portability.

UNIX may contain one disk driver to control all disk drives connected to the system and one terminal driver to control all terminals in the system. That is the one-to-one relationship between device drivers and device types. The device driver should distinguish among many devices it controls, that is, output for one terminal must not be sent to another.

In other words, associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a major device number that serves to identify it and is related to the device type. If a driver controls several disks of the same type, each disk has a minor device number that identifies it and is used to distinguish it among devices of one device type. Together, the major and minor device numbers uniquely identify a single I/O device.

The driver interface that the kernel builds between the file system , and the device drivers includes two tables: the block device switch table (bdevsw) and character device switch table (cdevsw). The former is for block special files; the latter for character special files. Figure 7.13 shows the driver interface between the file system and the device drivers. Figure 7.14 displays the simplified tables of bdevsw and cdevsw.

For system calls that use file descriptors (such as read and write system calls), the kernel follows pointers from the user file descriptor to the kernel file table and inode, where it checks out the file type. From the inode, it also gets the major and minor numbers. With the file type, the kernel knows which table (the bdevsw or cdevsw table) should be accessed. Furthermore, the kernel uses the major number as an index into the appropriate table (bdevsw or cdevsw), and calls the corresponding driver according to the system call being made, passing the minor number as a parameter.

In the user view, when a user accesses a special file through system calls, the file system examines whether it is a block special file or a character special file and determines its major and minor device numbers. The major device number is used to index into the entry of the corresponding table, bdevsw table for block special files or cdevsw table for character special files. The entry located contains pointers to the programs to call for opening the

device, reading the device, writing the device, and so on. The minor device number that tells which unit of this type of devices is chosen is passed as a parameter.

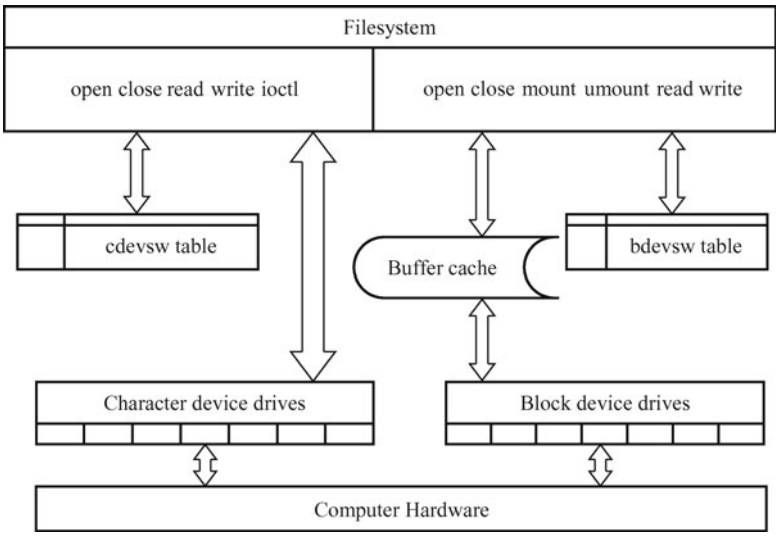

Fig. 7.13 The UNIX interface between the file system and the device drivers.

	Block device switch table		
	open	close	strategy
0			
1			
2			
3			

	Character device switch table				
	open	close	read	write	ioctl
0					
1					
2					
3					

Fig. 7.14 Simplified tables of bdevsw and cdevsw.

Also in UNIX, to add a new device type to the system is just to insert a new entry in one of these tables and to supply the corresponding programs to handle the various operations on the device.

Each driver is divided into two parts. One part runs in user mode with interfaces to the UNIX kernel. The other one runs in kernel mode and interacts with the device. Drivers are allowed to make calls to kernel processes for other operations, such as memory allocation, timer management, DMA control, and so on.

In classical UNIX operating systems, UNIX device drivers had been statically linked into the kernel, so they were all present in memory when the system was booted every time, which was really inflexible.

7.9.2 Block Special Files and Buffer Cache

As the disk special files are the main part of the block special files, the disk drivers will be paid more attention to. The disk driver converts a file system address, including a logical device number and block number, into a particular sector on the disk.

Since I/O operations involve in many mechanical actions and those actions take a lot of time (Bach 2006; Carson et al 1992; Heindel et al 1995; Quarterman et al 1985; Stallings 1998), the goal of block special files is to minimize the number of actual transfers between the memory and I/O devices. To accomplish this goal, UNIX uses a buffer cache between the disk drivers and the file system. On the other hand, system files, commands, and directories are usually reused frequently. If their data blocks are in the buffer cache, it is fast and effective for the kernel to find them directly in the cache when they are referenced because it saves to retrieve them from the disk and makes the cache hit rate high (see Figure 7.13).

Because the accessed block may be or not be in the buffer cache, the driver can get the address in one of two ways. One is that the strategy procedure (see Figure 7.14) uses a buffer from the buffer pool and the buffer header contains the device and block number; the other is that the read and write system calls are passed the logical (minor) device number as a parameter, with which they convert the byte offset saved in the user structure to the appropriate block address. The disk driver uses the device number to identify the physical drive and particular section to be used.

In the UNIX kernel, the strategy system call (see Figure 7.14) is used to transmit data between the buffer cache and a disk device. The strategy system call can queue I/O jobs for a device on a work list or do even more jobs. Drivers can set up data transmission for one physical address or more. The kernel passes a buffer header address to the driver strategy program; the header contains a list of addresses and sizes for transmission of data from or to the device. To the buffer cache, the kernel transmits data from one data address; when swapping, the kernel transmits data from many data addresses. We now discuss buffer headers in detail.

7.9.2.1 Buffer Headers and Lists

The buffer cache consists of many buffers that are managed in several different lists according to their used statuses. Each buffer is identified with its buffer header. The buffer header holds the following information:

- The device number and block number, where the block of a file that occupies the buffer now is on the disk.
- A pointer to where the buffer is located in the physical memory.
- The size of the location of the buffer in the physical memory.
- The memory amount in the location of the buffer that contains the file data.
- The dirty bit, which marks if the contents of the buffer element are mod-

ified.

When the system is booted, chucks of memory pages are specified as the buffer pool. Each page in the buffer pool is initiated as a buffer with a buffer header linked in a list. The size of the buffer pool and the number of buffer headers are relied on the available space in the primary memory.

The buffer headers are used to link the buffers of the buffer cache together so that the kernel can manage them easily. There are usually several different lists to link the buffers.

- Reserved region, which contains blocks that hold the resident part of the system that the kernel accesses frequently.
- Cache list, which holds blocks that are referenced recently and may be reused in the future.
- Assignable list, which collects blocks that have a lower possibility to be reused than ones in the cache list.

The cache and assignable lists are manipulated in the least recently used mechanism. The buffer headers are also hashed with the device and block numbers in the way like the one used in the hash frame queues of the memory management (see Section 5.3.1.2), which can speed up searching a buffer.

Except the above lists, some buffer headers can be empty and without disk blocks associated with yet, which are put in the free list and can be used to make up the buffer insufficiency when needed. And also, some buffers can be in process when some block device operations are being done. These buffers are not in any of the above lists, but in some waiting queue related to some block device.

Since in BSD there are two units of the data blocks on the disk (see Section 6.5.1.2): blocks and fragments, the maximum size of a buffer can be a block size (typically 8192 bytes), and the minimum one is a fragment size (1024 bytes). A buffer header can tell the buffer size, which may be a block size or less-than-eight contiguous fragments. That is, the buffer size can be varied. When a buffer is allocated to a block of a file and found that it cannot fit in the size of the block of the file, some appropriate free buffer has to be taken from the free list, and the previously-allocated buffer will be free and put on the free list.

The free list can also be complemented by the truncate system call, which can be used to reduce the size of a file. When the size of a file is reduced, the buffer that the file occupies will be decreased. For this reason, the kernel will allocate a smaller but appropriate free buffer to the file, and free the older one onto the free list.

7.9.2.2 Read from and Write to Disk Block

When a user process starts to read a file, the kernel first transfers a block on the file system in the buffer cache, and then copies the buffer to the user process's memory space. Similarly, when a user process begins to write a file, the kernel allocates a buffer to the process, and then the user process's

writing data are copied from the user process's memory space into the buffer. Both read and write are not done on the disk directly.

Therefore, when a process reads a block from a disk, the kernel first looks into the cache list to see if or not the block is there. If so, it is taken from there and a disk access is saved. In this way, the buffer cache greatly improves I/O performance.

If the block is not in the buffer cache, the kernel removes a buffer from the assignable list, updates its buffer header with the device number and block number of the block, reads the contents of the block first from the disk into the buffer, and copies from the buffer to where it is needed. If no buffers are in the assignable list, one buffer on the cache list can be chosen to allocate according to the least recently used mechanism. And if the chosen buffer is dirty, it has to write to the disk before being replaced with the new contents. With the least recently used mechanism, whenever a buffer is accessed, it is moved to the head of the list. When a block must be removed from the cache to make room for a new block, the one at the end of the chain is selected.

When the end of a buffer is referenced, the buffer is put on the assignable list because it is assumed that if the reference reaches the end of one block of a file, which is in the buffer now, the next reference may start at the following block of the file more possibly, which may cause a new block to be read in the buffer cache. If the end of the buffer has not been referenced, the buffer is more likely reused in the future so that it is still in the cache list.

The buffer cache also works for writes. When a process writes a block, it writes to the cache, not to the disk, temporarily. And the kernel marks the dirty bit of its buffer header. When the cache fills up, some blocks in the buffer cache must be forced out. If it is marked dirty, the buffer should be written to the disk. So it is added to the queue waiting for the disk I/O and the buffer is put on the assignable list. It is called the delayed write mechanism (Carson et al 1992; Ritchie et al 1974; Stallings 1998).

For the read, the temporary substitution of a disk block operation with a buffer operation cannot make any serious problems when the system crashes. However, for the write, the modified data in the buffer may not be really written to the file system on the disk if the system crashes unexpectedly, which can result in the data in the buffer lost and the data on the disk incorrect. To avoid this problem, the sync system call is periodically invoked to do write back to the disk — all the dirty blocks are usually written to the disk every 30 seconds. If all the blocks of a file have to be written to the disk at a time, for instance, in the situation for high data consistency, the fsync system call can replace the sync to do the write out.

Since the incorrect data in a directory, in-core inode or superblock can make even more serious problem, in UNIX, the kernel writes through to the disk each time it writes a directory, inode, or superblock in the buffer cache.

7.9.2.3 Character Device Drive for Disk

We know the block devices are structured, which are called structured I/O more properly. Therefore, the rest part of devices in the I/O system belongs to unstructured I/O. In UNIX, the devices in the system are divided into two groups: block devices and character blocks. Thus, except the block devices introduced above, all the others in the system are incorporated into the character devices.

Since many of the read and write operations are done by the interaction between the user and the system through the user interface of UNIX, which is a terminal or terminal window, the disk manipulating implementation inescapably involves the character devices and the character device files. In fact, the whole disk manipulating implementation includes three parts: a character device drive, a block device drive, and a swap device drive. The block device drive has been discussed in the last section, and the swapping has been introduced in Section 5.2. Thus, here we discuss the character device drive for the disk. In BSD versions, sometimes the character device drive is called raw device interface (Quarterman et al 1985).

In UNIX, disk manipulation is implemented via a queue of operation notes, which are called transaction records. Each transaction record holds the following fields:

- A flag, which tells whether the record is for reading or writing.
- A memory address, where the transfer is in the primary memory.
- A disk address, where the transfer is on the disk.
- The byte number of the transfer, which indicates the number of the transfer in bytes.

The character device drive for disk makes the transaction records according to the user process requests. It actually does the mapping of the transferred data from the user process memory space to the buffer. According to the transaction records, the block device drive accomplishes to transfer the data between the buffer and the disk, and makes its best to sort the transaction records in order to enhance the disk I/O operations. According to a transaction record, the swap device drive can do swapping if necessary.

7.9.3 Character Special Files and Streams

As the terminal special files are the main part of the character special files, the terminal drivers will be paid more attention to in this section. Terminals are special in the feature that they are the user hardware interface to the computer system. Since character special files are usually used for devices with which human beings interact with the computer system directly and deal with character streams, they do not move chunks of data between memory and disk. Therefore, the buffer cache is not needed for character special files,

as shown in Figure 7.13.

In fact, basically, there are two solutions to character drivers.

7.9.3.1 Line Discipline Solution

This solution was originated from BSD (Bach 2006; McKusick et al 2005).

To do interactive utility, terminal drivers have an internal interface to line discipline modules that interpret input and output.

For the internal interface, there are two modes that can be chosen: one is a raw mode, the other is a canonical mode. In canonical mode, the line discipline converts the raw character sequence typed at the keyboard to a processed character sequence that a process can recognize before sending the data to a receiving process; the line discipline also converts raw output sequences written by a process to a format that the human being is familiar with. In raw mode, the line discipline passes data between processes and the terminal directly without any interpretation. The raw mode is especially useful when sending binary data to other computers over a serial line or for GUIs. The main functions of a line discipline are listed in Table 7.2.

Table 7.2 Main functions of the line discipline

Function	Description
To receive the input	In a canonical mode, to generate signals to processes for terminal hangup, or in response to a user pressing control characters, such as the Delete, CTRL-S, or Enter key In a raw mode, to forbid interpreting special characters such as erase (Backspace or Delete), kill (CTRL-U) or carriage return (Enter)
To process	To parse input strings from the keyboard into lines To process erase characters (such as Backspace and Delete) To process a kill character (such as CTRL-U) that invalidates all characters typed so far on the current line
To send the output	In a canonical mode, to echo received characters to the terminal screen with interpretation In a raw mode, to echo received characters to the terminal screen without interpretation

The data structures handled by line disciplines are called c-lists, which represent character lists. A c-list is a linked list of cblocks with a count of the number of characters on the list. A cblock contains three fields: a pointer to the next cblock on the linked list, a small data block, and two offsets indicating the position of the valid data in the cblock. The start offset indicates the first location of valid data in the data block, and the end offset is the first location of non-valid data.

Clists provide a simple buffer holding the small volume of data that is fit for transmission of slow devices such as terminals. They allow dealing with one character at a time or groups of cblocks.

For terminal drivers, there are three clists associated with them: one is used to store data for output to the terminal screen, the second one is to store the raw input data provided by the terminal interrupt handler as the

user typed it in, and a third one is to store the processed input data that the line discipline converts the data of the raw clist into.

Therefore, when a user process reads from /dev/tty, the characters pass through the line discipline. The line discipline accepts the raw character sequence from the terminal driver, "cooking" it, and producing the cooked character sequence. The cooked sequence is passed to the process. However, if the process wants to interact on every character, it can put the line in raw mode, in which case the raw character sequence will be passed to the process directly without being cooked.

Output works in a reverse way, such as, converting a line feed to a carriage return plus a line feed, adding filler characters following carriage returns for slow mechanical terminals, and so on. Like input, output can go through the line discipline in the raw mode or canonical mode.

7.9.3.2 Streams Solution

This solution was from System V, and devised by Dennis Ritchie (Bach 2006; Ritchie 1984). It was issued at the background of some drawbacks of device drivers. That is, different drivers tended to duplicate functionality, such as network drivers which include a device-dependent portion and a protocol portion, but it was difficult to realize in the reality because the kernel did not provide the mechanisms for common use. Therefore, to get more flexibility and greater modularity, it is a mechanism to break down the modularity of the UNIX I/O system. That is Ritchie's solution.

The basic idea of a stream is to break down a whole device drive into several modules connecting from a user process to a driver and be able to insert processing modules into the stream dynamically for a certain utility. So a stream functions like pipelines in user space.

A stream is a duplex, multi-joint connection between a process and a device driver. It consists of a set of linearly linked queue pairs. Each queue pair has two members: one is for input, the other for output. When a process reads data from a stream, the kernel transfers the input data from the device driver up to the process through the input queues. In reverse, when a process writes data to a stream, the kernel sends the data down the output queues. The queues also pass messages to neighboring queue by an appropriate interface. Each queue pair belongs to an instance of a specific kernel module, such as a driver, line discipline, or protocol. Modules manipulate data passed through their queues. The stream infrastructure in the UNIX kernel defines the interfaces of modules so well that different modules can be plugged together.

A typical queue is a data structure that contains some elements shown in Table 7.3.

Table 7.3 Main elements of a stream queue

Element	Description
procedures	An open procedure that is called during an open system call
	A close procedure that is called during a close system call
	A put procedure that is called to pass a message into the queue
	A service procedure that is called when a queue is scheduled to execute
pointers	A pointer that points to the next queue in the stream
	A pointer that points to a list of messages awaiting service
	A pointer that points to a private data structure that maintains the state of the queue
flags and marks	Flags and high- and low-water marks that are used to control the data flow, schedule and maintain the queue state

A device with a stream driver is usually a character device that has a special field in the cdevsw table that points to a stream initialization structure, containing the addresses of procedures and high- and low-water marks shown in Table 7.3. For the first open system call of a stream driver, the kernel allocates two pairs of queues, one is for the stream-head, and the other is for the drive. The stream-head module is identical for all instances of open streams. It has generic put and service procedures and is the interface to higher-lever kernel modules that implement the read, write, and other system calls. Many modules, if needed, can be inserted into a stream by issuing ioctl system calls (see Figure 7.14). As queue pairs are bidirectional, each module maintains a read queue and a write queue; one is for reading, the other for writing. When

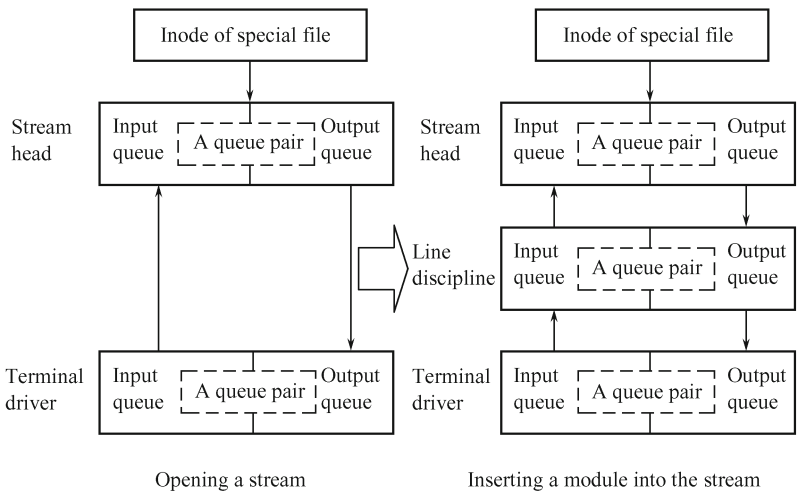

Fig. 7.15 A simple stream being created.

a user process reads from a stream, the stream head interprets the system call and packs the data into stream buffers that are passed from module to module upward. With each module performing their part in the transformation job correctly, the user process can read the data that it wants. A simple stream being created is displayed in Figure 7.15.

7.9.4 Sockets for Networks in UNIX

Networks are another type of I/O devices in UNIX. They are usually treated as sockets, which were started from BSD versions (Bach 2006). Sockets are used to handle network utilities, such as mailboxes, remote login, and remote file transfer. As shown in Section 4.5.3, in UNIX, there are many methods to maneuver inter-process communication even on different machines. Sockets can make processes communicate with other processes on other computers according to protocols and media.

For the socket mechanism, the kernel structure has three parts, which are shown in the order from the higher level down to the lower level in Table 7.4.

Table 7.4 Kernel structure for the socket mechanism

Layer	Function
Socket layer	To provide the interface between the system calls and the lower layers
Protocol layer	To contain the protocol modules used for communication (such as TCP and IP)
Device layer	To contain the device drivers that control the network devices (such as the network adapter card)

Appropriate combinations of protocols and drivers should be defined in the system configuration. Processes communicate using the client-server model (which will be discussed in Chapter 11), by which a server process listens to a socket at one end of a bidirectional communication path, and client processes communicate to the server process by another socket on another machine at the other end of the communication path. The kernel handles internal connections and routes data from client to server.

The socket mechanism involves several system calls. The socket system call establishes the end point of the communication link with the type of communication and the protocol to control the communication and returns a socket descriptor. With the socket descriptor, the bind system call associates an address that points to a structure holding the information of the communication domain and protocol. Before a socket can be used for networking, it must have an address bound to it. The address can be in one of several naming domains. The most common domain is the Internet naming domain. The connect system call sets up a connection for an existing socket. The listen

system call sets the queue length. The accept call allows incoming requests for a connection to a server process. The send and recv system calls send and receive data through a connected socket. The shutdown system call closes a socket connection. And so on.

Each socket supports a certain type of communication properties and follows a certain protocol. Here, two common types are listed:

- Virtual circuit type (or reliable connection-oriented byte stream): This socket type allows two processes on different machines to establish the equivalent of a pipe between them. Byte stream is sent in one end of the pipe and comes out in the same order at the other end. The system guarantees that all bytes sent out can reliably arrive at the receiver end.
- Datagram type (or unreliable packet-oriented transmission): This is especially useful for real-time utilities (such as, on-line movies). This type provides the unduplicated delivery, that is, it does not guarantee sequenced and reliable delivery. Packets sent out may be lost or reordered by the network. In some situation, higher performance is more important than reliability (for example, for multimedia delivery, fast transmission is more important than being correct).

The most popular protocol for virtual circuit type is Transmission Control Protocol (TCP). The common choice for the datagram type is User Datagram Protocol (UDP). Both protocols are basic for the Internet. We will discuss them in Chapter 11.

After sockets have been created on two computers, a connection can be established between them (for reliable connection). One side makes a listen system call on the server socket, which creates a buffer and blocks until data arrive. The other side makes a connect system call on the client socket, providing parameters such as the file descriptor for the client socket and the address of the server socket. If the server socket accepts the call, the system then establishes a connection between the two sockets. Once a connection established, it functions like a pipe. A process can read from and write to it using the file descriptor for its client socket. When the communication is no longer needed, it can be closed with the close system call.

7.10 Summary

Because of the mechanisms that UNIX uses to handle I/O devices, standard files, I/O redirections, and pipes and filters are closer to the file system in UNIX. Input redirection means to let a command input direct to a file rather than to standard input – a terminal keyboard. There are two different ways to do input redirection. One is to use the less-than symbol (<); the other is to use the < combined with the file descriptor 0 of the standard input file, stdin. Output redirection means to let a command output direct to a file rather than to standard output – a terminal screen. There are also two

different ways to do output redirection. One is to use the greater-than symbol (>); the other is to use the > combined with the file descriptor 1 of the standard output file, stdout. Expecting not to overwrite the previous contents of the destination files, use the appending redirection operator >>. As output redirection, standard error redirection means to let the error caused by the command execution direct to a file rather than to standard output – a terminal screen. The standard error is redirected only by using the > operator combined with the file descriptor 2 of the standard error file, stderr.

In the C shell, the input, output, and appending redirection operators (<, > and >>) do work as well as they do in other shells, but there is not an operator to redirect the stderr alone. The >& operator can be used for both the output and error redirections at the same time.

In most UNIX shells, the input, output, or error messages of a command can be redirected in one command line. The appending redirection >> operators can also be combined together with themselves or other redirection operators in one command line.

In UNIX, pipes and filters are usually used to perform some complicated tasks that cannot be done with a single command. The pipe operator (a vertical bar |) can be used to make the standard output of the command to the left of the pipe operator become the standard input of the command to the right of the pipe operator. If a command takes its input from another command, does some operation on that input, and writes the result to the standard output, the command is called a filter. Pipes and I/O redirections can be used in one command line.

In the C shell, the | operator can connect the output of one command to the input of another one. Otherwise, it also allows the output and error messages of one command to be attached to the input of another command with the |& operator.

Processes that are communicating with pipes usually have the same parent process while processes that are communicating with named pipes can be independently on one system. UNIX provides the mkfifo command to create named pipes. A named pipe can be removed from the file system by using the rm command, when it is no longer used.

In UNIX, the I/O devices are integrated into the file system, so the user interface to devices needs to go through the file system to control the I/O devices. That is, it is through a special file to access one of hardware devices, including block devices and character devices. To access a device, use the command or system call that accesses its special file. All I/O devices in UNIX are treated as files and are accessed as such with the same read and write system calls that are used to access all ordinary files. The driver interface that the kernel builds between the file system and the device drivers includes two tables: the block device switch table (bdevsw) and character device switch table (cdevsw). The former is for block special files; the latter is for character special files. A block special file consists of sequence of numbered blocks; the key property of the block special file is that every block can be accessed

with its own address. Character special files are usually used for devices with which human beings interact with the computer system directly.

To minimize the number of actual transfers between the CPU and I/O devices, UNIX uses a buffer cache between the disk drivers and the file system. The strategy system call can be used to transmit data between the buffer cache and a device. Usually, the buffers in the cache are linked together in a list, which adopts the least recently used mechanism to manage its buffers. When the cache fills up, some block in the buffer must be forced out. In UNIX, it is periodic to do write all the modified blocks back to the disk.

Actually, the whole disk manipulating implementation includes a character device drive, a block device drive, and a swap device drive.

There are two basic solutions to character drivers. One is the line discipline solution that was from BSD. For this solution, terminal drivers have an internal interface to line discipline modules that interpret input and output. The other is the streams solution that was from the System V. It is to break down a whole device drive into several modules connecting from a user process to a driver and be able to insert processing modules into the stream dynamically for a certain utility. A stream is a duplex, multi-joint connection between a process and a device driver. Modules have well-defined interfaces, defined by the streams infrastructure in the UNIX kernel, so different modules can be plugged together.

Networks are another type of I/O devices in UNIX. They are usually treated as sockets, which were started from BSD. Sockets are used to handle network utilities. Sockets can make processes communicate with other processes on other computers according to protocols and media.

Problems

Problem 7.1 What are standard files? State their purposes, respectively.

Problem 7.2 Describe input, output, and error redirection briefly. Write two commands of each to show single and combined use of the redirection operators.

Problem 7.3 How can the input, output, and error redirection operators be combined with the file descriptors of standard files to perform redirection in the Bourne shell? Give your examples.

Problem 7.4 Give a command sequence to create a short text file.

Problem 7.5 Give a command line to combine the pipe and output redirection operators.

Problem 7.6 What is the purpose of the tee? Give an example to use the tee.

Problem 7.7 What is the UNIX pipe? What is a filter? How different is pipe from output redirection? Give an example to illustrate.

Problem 7.8 What is a named pipe? Write down a single command to

create three named pipes, called myfifo1, myfifo2, and myfifo3. Write a command sequence by using these named pipes.

Problem 7.9 In UNIX, how can a process access one of hardware devices? How many kinds are hardware devices in UNIX divided in? What are they?

Problem 7.10 What is a block special file? What is a character special file?

Problem 7.11 Why is the delayed write mechanism adopted in UNIX?

Problem 7.12 What does the driver interface that the kernel builds between the file system and the device drivers include? How do they work?

Problem 7.13 In UNIX, what is a buffer cache? What is the buffer cache used for? How is the buffer cache managed?

Problem 7.14 What does the strategy system call do?

Problem 7.15 In UNIX, how many solutions are there to character drivers? What are they? Please explain how they work, respectively.

Problem 7.16 What are sockets used for? The socket mechanism involves several system calls. Explain how they function.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Carson SD, Setia S (1992) Analysis of the periodic update write policy for disk cache. IEEE T Software Eng 18(1): 44 – 54

Heindel LE, Kasten VA (1995) Real-time UNIX application filestores. RTAS'95: First IEEE Real-time Technology and Applications Symposium, Chicago, Illinois, 15 – 17 May 1995, pp 44 – 45

Isaak J, Lohnson L (1998) POSIX/UNIX standards: Foundation for 21st century growth. IEEE Micro 18(4): 88, 87

Jespersen H (1995) POSIX retrospective. ACM, StandardView 3 (1): 2 – 10

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Nelson BL, Keezer WS, Schuppe TF (1996) A hybrid simulation-queuing module for modeling UNIX I/O in performance analysis. WSC'96: The 1996 IEEE Winter Simulation Conference, 8 – 11 December 1996, pp 1238 – 1246

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Ritchie DM (1984) A stream input output system. AT&T Bell Lab Tech J, 63(8) Part 2: 1897 – 1910

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing

Stallings W (1998) Operating systems: internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey